Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria dell'Informazione

Tesi di Laurea

ELaw: elaborazione automatica di atti notarili

Relatore:
Prof. Enoch Peserico
Stecchini Negri De Salvi

Correlatore:
Ing. Federica Bogo

Laureando:
Matteo Ceccarello

Anno Accademico 2010-2011

30 Settembre 2011

Sommario

Questa tesi descrive la struttura e la realizzazione di un'estensione di OpenOffice Writer per effettuare la ricerca di parole significative in un documento in ambito notarile. In questo lavoro, che ha come obiettivo finale quello di aumentare la produttività negli uffici degli studi legali, si utilizza un meccanismo di ricerca basato su espressioni regolari. Per aumentare la velocità di esecuzione, tale procedura sfrutta il fatto che il documento viene scritto durante la ricerca stessa. Inoltre è stato adottato un approccio modulare che rende il programma facilmente estensibile. Infine, esperimenti mostrano come siano stati raggiunti sia gli obiettivi di accuratezza che di velocità di esecuzione.

RINGRAZIAMENTI

Desidero ringraziare tutti coloro i quali hanno contribuito alla realizzazione di questa tesi.

Il professor Peserico per aver ideato il progetto e la dottoressa Federica Bogo per la costanza e la gentilezza con cui ci ha seguiti in questo lavoro.

Il mio collega Alessandro Secco, instancabile ed inesauribile, sul cui lavoro ho potuto sempre fare affidamento.

La mia amata Vera, che mi ha aiutato a scovare gli errori presenti nel testo.

I miei genitori, il mio amico Marco e tutti i miei amici per l'affetto e il supporto in questi anni di studio.

INDICE

1 INTRODUZIONE

1.1 ELAW: UNO STUDIO NOTARILE INFORMATIZZATO

Il progetto ELaw si rivolge agli uffici e in particolar modo agli studi notarili. In tali uffici viene utilizzato un gran numero di dispositivi, che varia dal PC al telefono, oltre che stampanti e chiavi per la firma digitale. Spesso la varietà di tali dispositivi pone dei problemi di integrazione che si riflettono in una difficoltà di manutenzione e in perdita di produttività. Anche restringendo il campo ai soli applicativi usati col computer si notano le stesse problematiche.

L'obiettivo del progetto ELaw è quello di produrre un sistema hardware e software per gli studi notarili integrato e scalabile. I vari componenti sono i seguenti:

- Un unico modello di computer, con sistema operativo Debian GNU/Linux opportunamente modificato e con installati i programmi elencati di seguito
  - Un editor di testo-word processor con funzionalità specifiche per gli studi notarili
  - Un software di telefonia sia voip che analogica, con la possibilità di fare da segreteria telefonica e da registratore
  - Un filesystem distribuito e un sistema di backup
  - Un browser web, con estensioni apposite per interagire con l'editor di testo
  - Un programma per l'utilizzo di chiavi per la firma digitale
- Un unico modello di modem per la connessione sia internet che telefonica
- Delle stampanti con inchiostro indelebile, testate per funzionare con il sistema operativo scelto

Il vantaggio di usare un unico modello per ogni componente, oltre a consentire di testare la funzionalità dello stesso e l'interoperabilità con gli altri, è di rendere il sistema modulare e scalabile. Se infatti si presentasse la necessità di aggiungere una nuova macchina, basterebbe connetterla alla rete interna, e similmente se una macchina già presente dovesse guastarsi, sarebbe sufficiente sostituirla con una identica, con tutto il software necessario già preinstallato e configurato.

Per ottenere questo i computer devono essere predisposti opportunamente, ad esempio il sistema di backup utilizzato deve essere distribuito su tutte le macchine, in modo tale da non dipendere da un server centrale che inevitabilmente sarebbe il punto debole della rete. In questa maniera l'intero ufficio presenta un notevole grado di robustezza.

In questo elaborato si tratta dello sviluppo dell'editor di testo. Attualmente i notai redigono gli atti tramite un editor di testo (che potrebbe essere Microsoft Word ad esempio) e successivamente devono utilizzare appositi programmi (disponibili solo per piattaforma Windows) per compilare dei form e produrre dei file (principalmente binari e XML) da inviare via internet agli enti competenti, ad esempio Agenzia delle Entrate e Registro Imprese. Tali programmi sono:

- Fedra Plus [1]: produce binari e XML da inviare al Registro delle Imprese [2].
- Unimod [3]: produce degli XML da inviare all'Agenzia del Territorio [4].
- Comunica[5]: unisce l'output dei due programmi precedenti in un pacchetto da inviare a Camere di Commercio, INPS, INAIL e Agenzia delle Entrate [6].

1.2 SCOPI DEL PROGETTO

Il progetto ELaw ha molteplici scopi, molti dei quali già descritti in 1.1. Più in particolare il sotto-progetto riguardante il writer ha come scopi principali:

1. Eseguire un'identificazione il più possibile accurata dei campi presenti nel testo
2. Memorizzare in maniera adeguata i campi trovati
3. Produrre in maniera semi-automatica i file di output da inviare agli enti competenti.
4. Integrarsi strettamente con l'editor di testo, fornendo un'interfaccia utente consistente

In particolare questo elaborato tratta della ricerca dei campi all'interno del documento, mentre della parte di memorizzazione degli stessi si è occupato lo studente Alessandro Secco. La creazione dei file di output è in fase di sviluppo.

1.3 LO STATO DELL'ARTE

Il problema della ricerca dei campi nel testo può anche essere visto come un tentativo di classificare le parole che vi compaiono. Di questo si occupano le ricerche riguardanti Information Retrieval, Information Extraction e Document Retrieval.

1 http://www.infocamere.it/software.htm
2 http://www.registroimprese.it
3 http://www.agenziaterritorio.it/?id=701
4 http://www.agenziaterritorio.it
5 http://www.registroimprese.it/dama/comc/comc/IT/cu/
6 http://www.agenziaentrate.gov.it

Figura 1.: Schema a blocchi di un tipico sistema di Information Extraction, (figura presa parzialmente da [6, ch. 3, pag. 80])

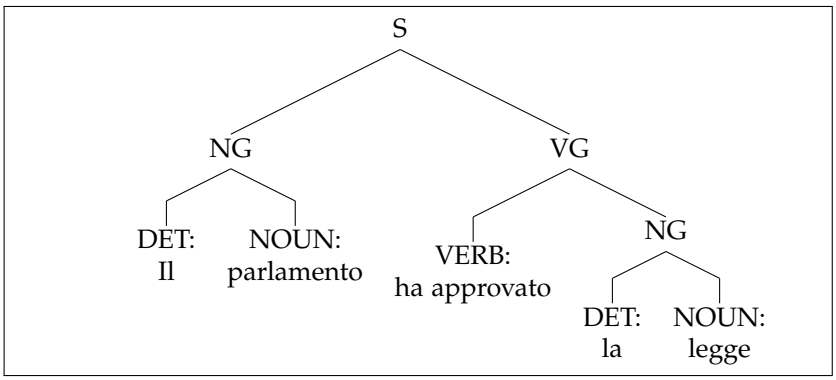

Figura 2.: Scomposizione della stringa "Il parlamento ha approvato la legge"

Vari approcci sono stati proposti per assolvere a questo compito. Uno di questi consiste nell'utilizzare le espressioni regolari [6, ch. 3]. Utilizzando questa metodologia si divide il testo in frasi, e si effettua la ricerca di strutture definite da espressioni regolari, come ad esempio (usando la notazione UNIX)

```
[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-Z]
```

che descrive i codici fiscali come `AAABBB11C22D334E`.

Tuttavia non tutte le parole possono essere classificate in questa maniera, quindi sempre in [6] viene posto in evidenza come si possa usare in congiunzione alle espressioni regolari un lessico e delle regole grammaticali per spezzare le frasi in gruppi di parole (Part Of Speech, POS) legati tra loro, ad esempio gruppi nome-attributo o verbo-avverbio e usare questa informazione aggiuntiva per meglio discernere fra loro le parole. Infatti la tipica struttura di un parser che fa uso di espressioni regolari è mostrata in fig. 1.

La divisione in POS può essere ottenuta grazie all'uso di Context Free Grammars per costruire una gerarchia delle parole delle strutture formate dalle parole all'interno della frase [6, par. 3.4]. Ad esempio la frase

Il parlamento ha approvato la legge.

sarà scomposta come in figura 2.

Le lettere maiuscole indicano le varie POS e le loro combinazioni:

S *Sentence*: la frase intera

NG *Noun Group*: un gruppo che aggiunge attributi a un sostantivo

VG *Verb Group*: comprende il verbo, gli avverbi e gli oggetti del verbo

VERB un verbo

NOUN un sostantivo

DET un articolo determinativo

Ovviamente occorre definire in maniera precisa le regole grammaticali per distinguere i vari gruppi, tenendo conto che molti casi sono ambigui, e vengono risolti dalle persone tramite il buonsenso e l'esperienza, doti di cui un computer purtroppo non è dotato. Tali regole possono essere definite manualmente, ma questo comporta un notevole lavoro.

L'alternativa alla definizione manuale delle regole è fare in modo che il programma le apprenda da solo a patire da un *training corpus*. Spesso si utilizza un approccio di tipo stocastico [6], ma un'alternativa più semplice è data in [1]. L'obiettivo è dividere la frase in gruppi racchiusi da parentesi (ma è solo una rappresentazione, è possibile anche usare un albero) che delimitano le POS, ad esempio

((Il parlamento) ((ha approvato) (la legge)).)

L'algoritmo si basa su un insieme limitato di operazioni (come aggiungere/togliere parentesi a destra/sinistra di un particolare POS) che vengono apprese da un training corpus già parentesizzato. Lo svantaggio di questi approcci che dividono il testo in POS risiede nel fatto che un requisito preliminare a questa divisione è l'analisi grammaticale (sia pur minima) unita a un lessico, in modo da riconoscere gli elementi di base, come nomi comuni, aggettivi, verbi transitivi e non e così via...

Un'altra possibile alternativa è data in [7], dove le parole estratte da un database precedentemente costruito vengono trovate e taggate nel testo. Ovviamente questo metodo è adatto al tagging automatico di pagine web (es. Wikipedia) dove mantenere una lista delle possibili *keyword* è un'opzione valida, ma non è applicabile al nostro caso, visto che, mentre è possibile avere una lista dei nomi propri italiani, non è pensabile mantenere un database di tutti i codici fiscali e partite IVA.

2 OPENOFFICE E L'ADDON ELAW

L'argomento trattato in questo capitolo è l'editor di testo scelto come base per il progetto e la struttura dell'addon di ELaw che aggiunge le funzionalità richieste. Prima vengono presentati gli addon di OpenOffice in generale (2.1), quindi viene descritto in dettaglio l'addon di ELaw (2.2).

Essendo impossibile per motivi di tempo iniziare la scrittura di un editor da zero, per il progetto si è dovuto scegliere un editor esistente da estendere. Tale programma deve avere alcune caratteristiche

- Essere WYSIWYG: non è infatti pensabile che il notaio editi un file LaTeXper ottenere la formattazione desiderata.
- Trattare testo formattato: probabilmente il notaio vorrà avere il controllo sul layout del documento (indentazione, rientri, allineamenti ecc.) quindi un semplice editor plain text (GEdit, Kate) non va bene.
- Essere estendibile: poter scrivere un'estensione senza toccare il codice sorgente originale del programma è enormemente più vantaggioso, oltre che più sicuro, visto che non si rischia di introdurre bug e instabilità nell'editor.
- Essere familiare all'utente: questo è un requisito meno stringente, tuttavia la maggior parte degli utenti finali probabilmente proviene da un'esperienza Windows, utilizzando Microsoft Office Word. Un editor che vi assomigli può semplificare molto l'esperienza dell'utente.

Più che un editor di testo quindi l'ideale è un word processor. Varie opzioni sono disponibili nel mondo Open Source: Openoffice.org, AbiWord, KOffice. KOffice (che fa parte della suite di software KDE) è stato scartato perché attualmente in fase di profondo rinnovamento, quindi non è garantita la continuità delle API fornite, e perché è strettamente integrato nel progetto KDE, richiedendo quindi l'installazione di numerose librerie estranee a GNOME, l'ambiente desktop scelto. Tra AbiWord e Openoffice la scelta è ricaduta su quest'ultimo per la sua diffusione, compatibilità con i formati proprietari (che quindi consente di leggere file già creati dal notaio) ed estensibilità. Durante il corso del progetto c'è stato il fork fra OpenOffice e LibreOffice, che per il momento tuttavia sono identici, quindi abbiamo preferito continuare ad utilizzare OpenOffice come piattaforma di base, per evitare un port inutile.

Si diceva che uno dei punti di forza della suite di OpenOffice è la sua estensibilità. Sono infatti disponibili e facilmente installabili svariati addon per assolvere ai più svariati compiti [1]. Dal punto di vista della

1 Molte estensioni possono essere scaricate a questo sito: `http://extensions.services.openoffice.org/`

programmazione gli addon possono essere scritti in diversi linguaggi di programmazione, fra i quali C++ e Java (cfr. [10]).

Il linguaggio scelto è Java, anche perché è il linguaggio con cui il gruppo di lavoro ha maggiore dimestichezza, per il quale la SDK di OpenOffice fornisce delle API per accedere alle varie parti dell'editor, tramite il framework UNO (Universal Network Objects).

Un difetto di OpenOffice è la scarsità di documentazione a disposizione dello sviluppatore di estensioni: le API sono molto scarne e la guida dello sviluppatore [10] è in alcune sezioni datata. Inoltre alcune funzionalità si trovano in uno stato indefinito: sono fornite da delle interfacce che secondo la documentazione sono deprecate (e che non funzionano) ma le interfacce e le classi che le devono sostituire non sono ancora disponibili, rendendo di fatto queste features (fra cui la possibilità di intercettare gli eventi di salvataggio del documento) inutilizzabili.

2.1 GLI ADDON DI OPENOFFICE

Un addon in OpenOffice è un'estensione che aggiunge nuove funzionalità, eventualmente integrandosi nell'interfaccia utente con nuovi menu e pulsanti.

2.1.1 Struttura e installazione del pacchetto di un addon

Gli addon vengono forniti in pacchetti (con estensione ".oxt") che vengono installati tramite il gestore apposito di OpenOffice. L'installazione può avvenire sia per via grafica ("Strumenti→GestioneEstensioni") sia da linea di comando tramite `unpkg add nomepacchetto.oxt`. In tal modo l'installazione è eseguibile anche da script.

Vediamo più nel dettaglio la struttura di un pacchetto. Si tratta semplicemente di un file zip, avente tuttavia estensione .oxt, che al suo interno contiene obbligatoriamente almeno i seguenti file e cartelle

- `description.xml` : contiene la descrizione del pacchetto
- `registry` : cartella contenente i file che regolano la creazione dell'UI dell'addon e le interazioni con OpenOffice
  - `Addons.xcu` : descrive in formato XML le caratteristiche dell'interfaccia grafica dell'addon
  - `ProtocolHandler.xcu` : definisce i nomi dei protocolli con cui Openoffice interagirà con l'addon
- `META-INF` : contiene il file
  - `manifest.xml` : enumera i file del pacchetto che devono essere utilizzati durante l'installazione dell'addon

Oltre a questi possono essere inclusi nel pacchetto anche altri file, come ovviamente il codice che implementa l'addon e le immagini da associare ai bottoni.

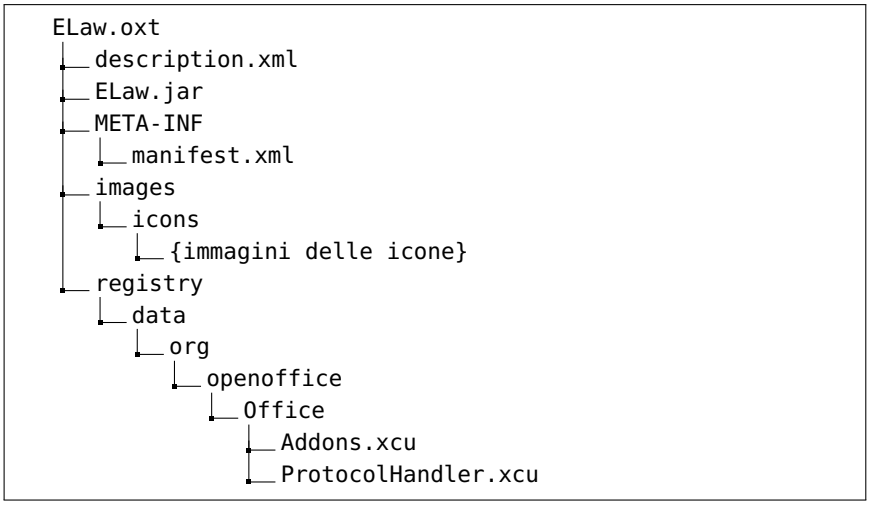

Figura 3.: Struttura del pacchetto .oxt dell'addon ELaw.

2.2 L'ADDON ELAW

L'obiettivo dell'addon eLaw è di fornire all'utente che utilizza OpenOffice una serie di strumenti per gestire la ricerca dei campi significativi all'interno di un documento in via di stesura e la produzione dell'output da inviare agli enti competenti a partire dai campi trovati.

Come è stato detto nella sezione 2.1.1 l'estensione deve essere impacchettata in un file con estensione `.oxt`. Nella figura 3 sono mostrati i file principali inclusi.

Questa struttura è stata ricavata empiricamente, a causa della scarsità della documentazione reperibile sul packaging degli addon e delle estensioni [10, ch. 5], dal template usato dall'IDE Netbenas per creare estensioni per OpenOffice.

2.2.1 Architettura

L'estensione ha una struttura modulare, dove ogni modulo ha una funzione specifica, e tutto questo si riflette nei pacchetti in cui sono raggruppate le classi, elencati di seguito:

- `archive` : i campi trovati vengono memorizzati nell'archivio implementato in in questo pacchetto, che fornisce metodi per accedervi successivamente.
- `field_table` : si tratta di un'interfaccia utente (simile a un foglio di calcolo) che consente all'utilizzatore di inserire i campi eventualmente non trovati dal programma e di effettuare correzioni.
- `main` : il componente centrale dell'addon è contenuto qui. Si occupa di interfacciarsi con OpenOffice, inizializzare tutti gli altri moduli, farli comunicare con OpenOffice e fra loro.
- `search` : le classi che fanno parte di questo pacchetto implementano gli algoritmi di ricerca nel testo descritti in questo elaborato.

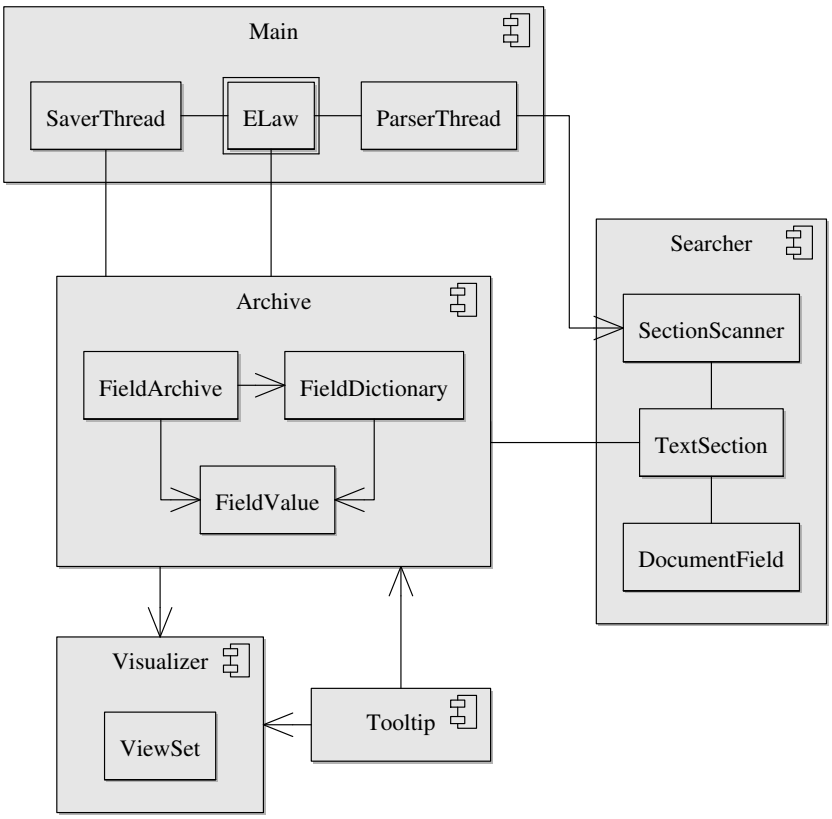

Figura 4.: Struttura e relazioni fra i moduli dell'addon

- `tooltip` : fa parte dell'interfaccia utente e permette di modificare i campi trovati, eliminarli e aggiungerli al volo senza passare da `field_table`, tramite un semplice <ctrl>+click
- `util` : varie classi d'utilità per il resto del programma sono contenute qui.
- `visualizer` : permette all'utente di scegliere fra due differenti modi di visualizzazione dei campi trovati, durante l'editing del documento: `label` che evidenzia il testo dei campi, `eagle` che proprone una visione d'insieme di tutti i campi associati alla loro chiave, e `quiet` che al contrario non mostra alcuna visualizzazione.

Attualmente sono in fase di sviluppo da parte di un altro studente i moduli per la produzione dei file di output.

La struttura e le relazioni fra i moduli sono mostrati in figura 4.

Il primo componente a essere inizializzato è la classe ELaw, che viene richiamata da OpenOffice al momento dell'avvio. Questa classe quindi inizializza l'archivio e due differenti thread: ParserThread e SaverThread. Il primo si occupa di scansionare periodicamente il documento tramite il componente Searcher (oggetto di questo elaborato), mentre il secondo si occupa del salvataggio dell'archivio su file. Come si vede, uno dei componenti centrali è proprio l'archivio (del quale si è

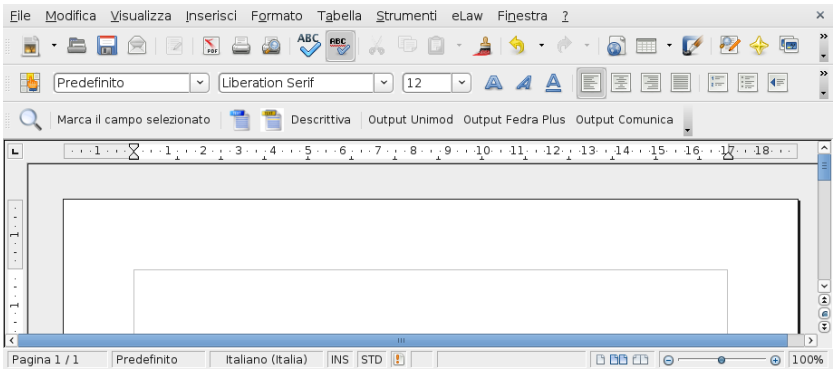

Figura 5.: Finestra di OpenOffice con addon ELaw integrato.

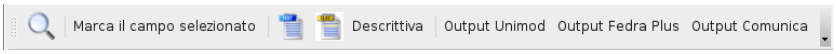

Figura 6.: La toolbar dell'addon eLaw

occupato lo studente Alessandro Secco), dal quale prendono informazioni i componenti di interfaccia utente di cui si parla nella prossima sezione: Visualizer e Tooltip.

I moduli per la creazione dell'output non sono ancora inclusi nel programma, ed è in corso di sviluppo l'archivio per creare l'output.

2.2.2 Aspetto e interfaccia utente

L'addon eLaw si integra con l'interfaccia utente di OpenOffice aggiungendo una toolbar e un menu a tendina (fig. 5). Nella toolbar sono presenti otto bottoni, replicati nel menu a tendina (fig. 6):

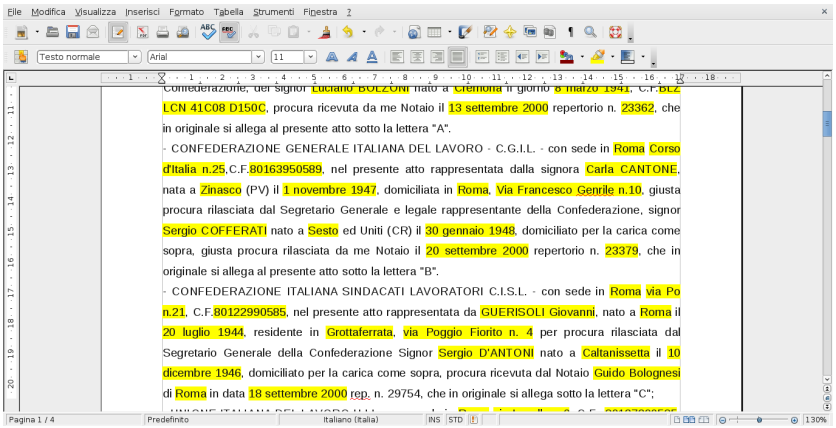

Figura 7.: La vista label: evidenzia tutti i campi trovati in modo da consentirne una rapida visualizzazione.

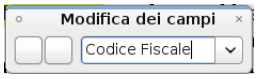

Figura 9.: La finestra di modifica di un campo

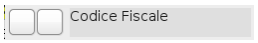

Figura 10.: La finestra informativa di un campo

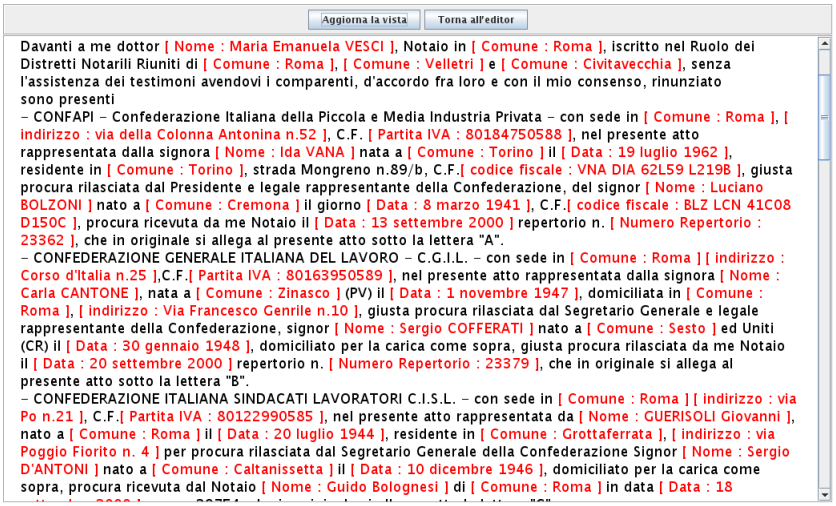

Figura 8.: La vista eagle: crea una nuova finestra con i campi nel formato [Nome Campo: valore campo]. Ancora sperimentale.

1. *Scansiona il documento* Avvia la scansione periodica del documento (si veda il capitolo 3).

2. *Modifica il campo selezionato* Richiama una finestrella di modifica per modificare il campo selezionato o per crearne uno nuovo (fig 9).

3. *Visualizzazione plain* Visualizza il documento come un normale file di OpenOffice, ovvero testo semplice con le formattazioni inserite dall'utente.

4. *Visualizzazione label* I campi significativi trovati vengono evidenziati con uno sfondo colorato, come in figura 7. In questa modalità tramite il click del mouse su una parola evidenziata, tenendo premuto il tasto ctrl, fa comparire una finestrella informativa (fig. 10) che consente anche di modificare il campo selezionato, tramite il primo bottone.

5. *Visualizzazione eagle* Apre una nuova finestra nella quale il testo del documento viene riproposto con i campi nella forma

   ```
   [tipo di campo : valore]
   ```

   quindi ad esempio un codice fiscale apparirà in questa maniera

```
[Codice Fiscale : AAABBB11C22D334E]
```

Un esempio è mostrato in figura 8.

6. *Output Unimod* Produce l'output Unimod.
7. *Output ComUnica* Produce l'output ComUnica.
8. *Output Fedra Plus* Produce l'output Fedra Plus.

3 LA RICERCA NEL TESTO

Questo capitolo tratta della ricerca dei campi nel testo da un punto di vista sia concettuale che della struttura delle classi utilizzate nell'implementazione. Dopo aver illustrato i concetti di base (3.1) si parla della classificazione dei campi (3.2) e della struttura delle classi usate per la ricerca (3.3. Infine in 3.4 sono descritte le funzionalità che saranno implementate in futuro.

3.1 CONCETTI DI BASE

Il contesto in cui si svolge la ricerca dei campi è quello di un documento non completo e in via di stesura. L'obiettivo , oltre all'accuratezza dei risultati, è anche quello dell'efficienza in questo contesto.

L'euristica che sta alla base della ricerca nel testo è che i campi sono confinati all'interno delle singole frasi. Quindi invece di effettuare la ricerca periodicamente su tutto il documento conviene spezzare lo stesso in più frammenti elementari (ovvero le singole frasi) e poi ricercare i campi di interesse solo in quelli che sono cambiati. In questo modo la ricerca viene fatta solo dove veramente necessario, garantendo l'efficienza nel contesto di un documento in via di modifica.

È necessario quindi definire anzitutto due cose

- una politica di scansione del documento
- una politica di divisione del testo e determinazione della modifica o meno dello stesso

3.1.1 Scansione del documento

Avendo a disposizione una lista delle sezioni in qui è diviso il testo è possibile iterare su queste sezioni, controllando di volta in volta se la sezione in esame è stata modificata dall'ultima scansione. In caso di risposta affermativa si procede quindi a cercare i campi di interesse all'interno di tale sezione, per poi procedere a quella successiva.

3.1.2 Determinazione delle sezioni e loro modifica

Il testo può essere diviso in sezioni corrispondenti alle frasi, quindi essere separate dai punti di fine frase. Quindi è necessario distinguere questi ultimi dai punti che non concludono una frase (come ad esempio "Sig." o "s.p.a.").

In questo elaborato ci si riferisce a "punto di fine frase" come al segno di punteggiatura che conclude una frase. Tali segni sono ". ! ?", con l'esclusione di alcuni casi particolari, quali punti che chiudo-

	Nome Campo
Regex Based	Codice Fiscale Partita IVA Indirizzo Data Metratura Codice IBAN Coordinate Bancarie e-mail Numero di telefono Numero carta di Identità Numero di Repertorio dell'atto Numero di Raccolta dell'atto
Set Based	Nome Cognome Provincia Città CAP

Tabella 1.: Tipi di campi riconosciuti (* = non ancora implementati)

no un'abbreviazione, che separano cifre di un numero decimale, che separano le lettere di una sigla.

Le sezioni vengono create a partire da un'unica sezione contenente tutto il testo. Se questa prima sezione contiene un punto di fine frase, si usa come posizione di "split" per creare due nuove sezioni separate, e così via. In tale maniera lo stesso procedimento può essere applicato sia per creare le nuove sezioni all'inizio sia per aggiornarle successivamente. Infatti supponiamo di aver effettuato delle modifiche a una sezione e di avervi inserito una nuova frase. In questo modo la sezione contiene due frasi. Quando viene interrogata dall'oggetto che effettua la scansione per sapere se sono state effettuate modifiche dall'ultima ricerca, la sezione (o meglio, l'oggetto che la rappresenta) oltre a rispondere di sì controllerà se sono presenti dei punti di fine frase, nel qual caso si dividerà in due e informerà l'oggetto scansionatore dell'avvenuta divisione in modo che quest'ultimo possa gestirla.

Dato che la sezione deve essere in grado di determinare se sono state effettuate modifiche dall'ultima ricerca, memorizza la stringa che è stata oggetto di tale ricerca. Quando viene interrogata riguardo a eventuali modifiche la sezione semplicemente farà un confronto fra la stringa memorizzata e la stringa attuale.

3.2 CLASSIFICAZIONE DEI CAMPI

Per poter essere trovati nel testo utilizzando le metodologie descritte in § 3.1 i campi sono divisi in varie tipologie:

- Campi che possono essere descritti da una espressione regolare
- Campi che fanno parte di un insieme predefinito di parole

I campi riconosciuti e la loro suddivisione sono riportati nella tabella tabella 1.

Spesso l'informazione fornita dall'espressione regolare che descrive solo il campo non è sufficiente per identificarlo all'interno di un testo, quindi in alcuni casi (ad esempio "1000 euro" in cui 1000 è chiaramente una somma di denaro, ma questa informazione ci viene data dalla parola "euro") è necessario includere nell'espressione anche elementi del contesto circostante, facendo poi uso delle caratteristiche delle espressioni regolari in Java per selezionare solo la porzione di testo di interesse.

3.2.1 Campi descritti da espressioni regolari

In questa categoria ricadono tutti quei campi che hanno una struttura ben definita, come i codici fiscali o gli indirizzi, che quindi possono essere descritti da un'espressione regolare. Ad esempio un codice fiscale viene riconosciuto attraverso l'espressione mostrata nel listing 1.

```
1 [A-Z]{3} [\ ]? [A-Z]{3} [\ ]?
2 [0-9]{2} [A-Z] [0-9]{2} [\ ]?
3 [A-Z] [0-9]{3} [A-Z]
```

Listing 1: Espressione regolare per identificare i codici fiscali. Gli a capo e gli spazi sono aggiunti solo per questione di leggibilità

Questa espressione permette di riconoscere

```
CCCMTT89H24G224F
```

ma anche

```
CCC MTT 89H24 G224F
```

che è un'altra forma molto diffusa con cui sono scritti i codici fiscali.

Un altro esempio di campi che vengono riconosciuti tramite espressioni regolari sono gli indirizzi, come mostrato nel listing 2.

```
1 ( [Vv]ia | [Pp]iazza | [Cc]orso )
2 ['\s\w^\d]*
3 [,\.]? [\s]?
4 [\d]+
```

Listing 2: Espression regolare che identifica gli indirizzi. Gli spazi sono inclusi per motivi i leggibilità

La riga 1 identifica quello che è l'inizio di un indirizzo, ovvero le parole "via", "piazza" e "corso" seguite da un numero arbitrario di parole e spazi, ma non numeri (riga 2). Il campo è chiuso da un segno di punteggiatura opzionale seguito da uno spazio anch'esso opzionale (riga 3) e da una o più cifre, ovvero il numero civico (riga 4).

I campi che ricadono in questa categoria vengono quindi ricercati attraverso espressioni regolari.

3.2.2 Campi costituiti da parole appartenenti a un insieme

Alcuni campi oltre a far parte un linguaggio regolare possiedono anche la caratteristica di poter assumere un numero finito di valori. Esempi di questo tipo sono dati dalle province, dai CAP o dai nomi propri delle persone.

1. Attraverso un'espressione regolare appropriata viene selezionato un candidato che come struttura potrebbe far parte dell'insieme (ad esempio per un nome proprio deve iniziare con una lettera maiuscola).

2. Le parole che formano la stringa candidata vengono testate per l'appartenenza all'insieme prima ad una a una, poi a coppie e così via. Se uno di questi test ha successo, allora la stringa che era stata selezionata al punto 1 viene riconosciuta come campo.

Ad esempio per il campo "nome", Ceccarello Matteo deve essere riconosciuto. Attraverso l'espressione regolare che descrive i gruppi di parole che iniziano con la maiuscola la stringa "Ceccarello Matteo" viene selezionata come candidato. Successivamente si verifica se "Ceccarello" appartiene all'insieme dei nomi conosciuti, ma dato che in memoria ci sono solo i nomi propri questo test fallisce. Si passa quindi alla parola successiva, "Matteo". Dato che appartiene alla lista dei nomi conosciuti, l'intera stringa Ceccarello Matteo viene selezionata come campo "nome".

In realtà l'espressione regolare è molto più complessa, dato che in un atto possono capitare frasi del tipo "Il Signor Mario Rossi" dove solo "Mario Rossi" va selezionato come candidato. Inoltre è consuetudine negli atti scrivere i cognomi (e talvolta che i nomi) con tutte le lettere maiuscole. L'espressione regolare di selezione deve tenere conto di tutti questi casi. Per i nomi è riportata nel listing 3.

```
(                              # Inizio gruppo 1
   (?: [A-Z][A-Za-z]+ \s )*
      (?:
         Signor | SIGNOR |
         Notaio | NOTAIO |
         Dottor(?:essa)? | DOTTOR(?:ESSA)? |
         Dott\.ssa | DOTT\.SSA |
         Io
      ) \s
   )*
   (                           # Inizio gruppo 2
      (?:  [A-Z][a-z]+  |  [A-Z]+   )
      (?:
         \s (?: \w' )?
         (?: [A-Z][a-z]+  |  [A-Z]+  )
   )*                          # Fine gruppo 2
)                              # Fine gruppo 1
```

Listing 3: espressione regolare per selezionare i candidati a essere identificati come nomi. Gli spazi sono solo per motivi di leggibilità. Viene utilizza la capacità di Java di selezionare dei sottogruppi. In questo caso viene selezionato il gruppo numero 2.

Dalla riga 2 alla 10 sono descritte le parole che iniziano con la maiuscola (o con tutti i caratteri maiuscoli) che sono seguite dalle parole elencate nelle righe da 4 a 8. Queste parole se presenti vengono escluse in quanto viene selezionato il gruppo 2, che identifica le parole che iniziano con la maiuscola o che son tutte in maiuscolo.

L'esempio è stato preso direttamente dal file di configurazione delle descrizioni dei campi, si veda l'appendice B.

3.3 LA STRUTTURA DELLE CLASSI

Per implementare il metodo di ricerca esposto nella sezione 3.1 è necessario definire due classi, `TextSection` per identificare le sezioni, `SectionScanner` per effettuare la ricerca attraverso le stesse. Le relazioni fra le classi sono esplicitate nel diagramma UML di figura 12.

Inoltre è necessario fornire una rappresentazione dei campi, che nasconda l'implementazione in modo da ottenere una maggiore estensibilità.

3.3.1 DocumentField: una classe per rappresentare un campo generico

L'oggetto che rappresenta i campi da identificare è `DocumentField`. Si tratta di una classe astratta che fornisce principalmente due metodi:

- `getName()`: restituisce il nome del campo sotto forma di stringa.
- `find(String)`: cerca tutte le occorrenze del campo all'interno della stringa passata come parametro e ritorna una lista concatenata di coppie, dove ogni coppia (rappresentata da un array con due posizioni) indica l'inizio e la fine di ogni occorrenza. Se il campo non è presente viene restituita al chiamante una lista vuota. `DocumentField` non fornisce un'implementazione per questo metodo, lasciando questo compito alle classi che la estendono.

Questi due metodi sono gli unici necessari a `SectionScanner` per compiere la ricerca, quindi l'utilizzo di questa classe astratta nasconde i dettagli implementativi del metodo find, aumentando l'estensibilità e la flessibilità del meccanismo di ricerca.

Come visto in § 3.2 i tipi di campi sono due e ognuno di essi ha una classe che lo rappresenta: i campi basati su espressioni regolari `RegexBasedDocumentField`, quelli associati a un insieme di parole `SetBasedDocumentField`. In figura 11 è riportato il diagramma UML delle relazioni fra queste classi.

3.3.2 RegexBasedDocumentField

Questa classe estende `DocumentField` fornendo la propria implementazione del metodo `find(String)` e realizzando il procedimento di ricerca descritto in 3.2.1. La ricerca è effettuata appoggiandosi alle classi del pacchetto java.util.regex. Fino a che ci sono sottostringhe del testo in input che corrispondono all'espressione regolare, gli indici di inizio e

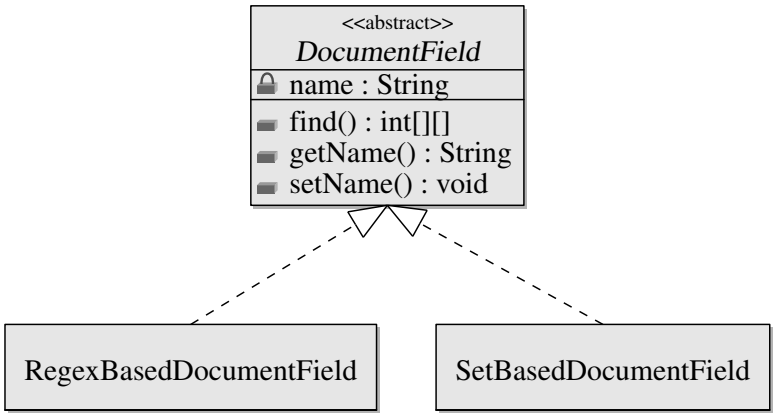

Figura 11.: Diagramma UML delle classi che descrivono i campi da trovare nel testo

fine di tali sottostringhe sono aggiunti alla lista che verrà ritornata alla fine della computazione.

3.3.3 SetBasedDocumentField

Questa classe effettua la ricerca in maniera simile a `RegexBasedDocumentField`, con l'aggiunta di un controllo di appartenenza ad un insieme. Tramite un'espressione regolare vengono selezionati dei candidati e successivamente le parole che li compongono vengono testate per l'appartenenza all'insieme associato al campo. Questa è la procedura di ricerca descritta in 3.2.2.

3.3.4 La classe `TextSection`

Questa classe rappresenta una sezione di testo, pertanto ha i seguenti attributi:

- `XTextRange` text: è il testo contenuto nella sezione
- `String` lastSearchedText: la stringa su cui è stata effettuata l'ultima ricerca

Inoltre alla sezione di testo è richiesto di fornire i seguenti metodi:

- `LinkedList<TextSection> split()` : divide la sezione in due, usando come punto di separazione il primo punto di fine frase trovato, e ritorna una lista contenente le due nuove sezioni
- `String freezeText()`: salva la stringa racchiusa all'interno della variabile `text` per utilizzarla in confronti successivi per determinare se ci sono state modifiche
- `boolean isModified()`: controlla se il testo contenuto nella sezione è stato modificato dall'ultima invocazione di `freezeText()`
- `boolean hasToBeSplit()`: controlla se nella sezione sono contenuti più punti di fine frase, nel qual caso la sezione deve essere divisa in due tramite il metodo `split()`

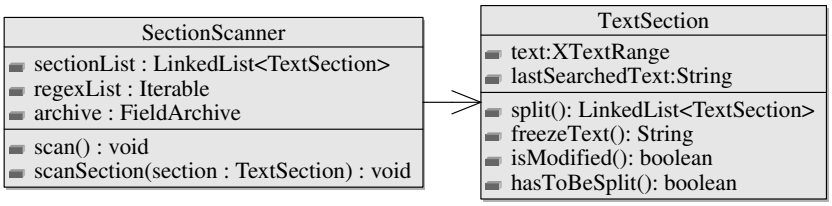

Figura 12.: Diagramma UML delle classi `SectionScanner` e `TextSection`

Questa classe e i suoi metodi verranno poi utilizzati dalla classe `SectionScanner` per compiere le ricerche.

3.3.5 La classe `SectionScanner`

Il compito di questa classe è principalmente quello di scorrere le sezioni in cui è diviso il testo controllando se ci sono state modifiche e in caso affermativo effettuare una ricerca di campi significativi.

Vengono utilizzate i seguenti campi d'esemplare:

- `LinkedList<TextSection> sectionList` : la lista delle sezioni di cui è composto il testo
- `FieldArchive archive` : l'archivio utilizzato per memorizzare i campi trovati

e i seguenti metodi per effettuare la ricerca:

- `void scan()` : scorre la lista delle sezioni, effettuando la ricerca in quelle modificate
- `void scanSection(TextSection section)` : ricerca i campi significativi nella sezione data, viene chiamato dal metodo `scan()`

3.4 SVILUPPI FUTURI

La struttura modulare del metodo di ricerca permette di estendere, oltre ai campi trovati, anche il meccanismo stesso della ricerca.

Infatti se si presentasse la necessità di aggiungere un campo che non è descrivibile né tramite espressione regolare, né tramite l'appartenenza a un insieme, invece di riscrivere completamente tutta la procedura di ricerca basterebbe scrivere una nuova classe che estende `DocumentField`. Questa nuova classe deve fornire il metodo `find()`, ma una volta fatto questo è facilmente inserita nel contesto generale senza modifiche al resto del codice.

Sono possibili anche estensioni in altre direzioni: attualmente sono in sviluppo un modulo per il riconoscimento automatico delle relazioni fra i campi trovati e un altro modulo per il learning automatico dei nuovi campi. I concetti di base di questi moduli sono esposti di seguito.

3.4.1 Riconoscimento delle relazioni fra i campi

Anche il riconoscimento delle relazioni fra i campi si basa sull'utilizzo delle espressioni regolari. Queste infatti presentano vari vantaggi: sono flessibili, consentono con semplicità di descrivere stringhe anche complesse e le API java ne forniscono un'implementazione robusta ed efficiente.

Attualmente il lavoro si concentra sul riconoscere le relazioni che intercorrono fra campi adiacenti, ovvero che si susseguono nel testo. Prendendo per esempio la frase

> ... il cliente **Mario Rossi**, nato a **Padova** il **01/01/1970** ...

il modulo di riconoscimento campi trova correttamente le parole in grassetto. Evidentemente questi tre campi si riferiscono tutti alla stessa persona, Mario Rossi, e quindi l'archivio deve esserne messo al corrente.[1]

Gli elementi fondamentali della ricerca delle relazioni sono due: i campi e gli "operatori di relazione". I campi sono quelli trovati dalla ricerca nel testo, mentre gli operatori di relazione sono quelle parole o gruppi di parole che semanticamente uniscono diversi campi. Esempi di questi operatori sono "nato a", "in provincia di", "in data" e simili.

Nella frase presa in esame precedentemente gli operatori di relazione sono sottolineati.

> ... il cliente **Mario Rossi**, nato a **Padova** il **01/01/1970** ...

Per trovare le relazioni per prima cosa si seleziona il testo fra due campi adiacenti e successivamente si tenta di fare il *match* con l'espressione regolare che descrive le relazioni. Se il match ha successo allora i due campi sono relazionati tra loro, quindi l'archivio ne viene informato, altrimenti sono due campi scorrelati.

Per i campi non adiacenti si sfrutta il fatto che le relazioni sono supposte transitive: se una persona è in relazione a una città (come nell'esempio) e tale città è in relazione a una data, allora la persona sarà in relazione a quella data.

3.4.2 Apprendimento automatico

Il meccanismo di riconoscimento, per quanto accurato, non può essere infallibile. È sempre possibile che un campo si presenti in una forma non prevista, o che nel documento ci sia un caso particolare. In queste occasioni l'utente ha la possibilità di inserire a mano il campo mancante evidenziando la parola e, tramite la combinazione <ctrl>+click o l'apposito pulsante nella barra di ELaw, far comparire il tooltip di inserimento/modifica. In questo modo può selezionare il nome corretto del campo da assegnare alla porzione di testo selezionata.

Tuttavia ripetere molte volte questa procedura per lo stesso campo, oltre che su documenti diversi anche sullo stesso documento, può essere frustrante. Quindi si rende necessario un meccanismo di apprendimento automatico di nuovi campi, che nella sua implementazione più semplice si presenta come una lista di casi particolari.

1 Della memorizzazione delle relazioni si occupa la classe `archive.InterconnectedFieldValues` sviluppata dallo studente Alessandro Secco

Per questo si utilizza un ulteriore file di configurazione, che tuttavia al momento dell'installazione dell'estensione è vuoto, per tenere traccia di tutti i casi particolari. Infatti verrà aggiornato dal programma stesso ogni volta che l'utente inserirà un nuovo campo manualmente. La struttura di questo file è a coppie chiave - valore, dove la chiave è la stringa che costituisce il caso particolare a cui è associato il nome del campo relativo come valore. Il contenuto di questo file sarà incluso nella procedura di ricerca: prima di procedere con la scansione della sezione descritta in 3.1 si cercano nella sezione le stringhe dei casi particolari, marcandole con l'etichetta del campo ad esse associato.

Ad esempio durante la stesura del documento il notaio si accorge che il nome "Annamaria" non è stato riconosciuto come tale (probabilmente perché non presente nel file `names.dat` che raccoglie tutti i nomi conosciuti dal programma). Inoltre anche "Arzercavalli" non è stata riconosciuta come città. Il notaio quindi tramite la finestra di inserimento/modifica campi li marca come nome e città. Quest'azione ha come effetto, oltre all'inserimento nell'archivio dei campi, anche l'aggiunta in coda al file dei casi particolari di queste due righe:

```
1 Annamaria : Nome
2 Arzercavalli : Citta'
```

Una volta inseriti nel file, questi casi particolari saranno disponibili per le scansioni successive, anche su documenti diversi.

4 ALGORITMI PER LA RICERCA

In questo capitolo si parla degli algoritmi utilizzati nella ricerca dei campi all'interno del testo e implementati nelle classi del pacchetto `search`. Sono presentati separatamente i metodi di `SectionScanner` (4.1), e di `TextSection` (4.2):

L'idea di base è la seguente: il metodo `scan` scorre la lista delle sezioni e per ogni sezione controlla se sono state fatte modifiche. Oltre a questo controllo tramite il metodo `findSplitIndex` si assicura che la sezione non contenga punti di fine frase, in caso contrario la divide in due tramite il metodo `split` della sezione stessa. Se risultano essere state fatte modifiche alla sezione invoca il metodo `scanSection`.

4.1 METODI DELLA CLASSE `sectionscanner`

Di seguito sono descritti gli algoritmi impiegati dalla classe SectionScanner per portare a termine il compito di scansionare il documento in via di stesura.

4.1.1 Il metodo `scan`

Questo metodo (algoritmo 1) utilizza la lista concatenata `sectionList`, campo d'esemplare della classe `SectionScanner`. Per questioni implementative (è necessario evitare il lancio di una eccezione dovuta alla modifica della lista da parte di due thread separati, ovvero l'iteratore del ciclo for e il codice contenuto nel ciclo), bisogna usare una lista di supporto, inizializzata alla riga 2.

L'algoritmo prende la prima sezione della lista e controlla se deve essere divisa; se la risposta è affermativa la divide, ottenendo due `TextSection`. La seconda viene aggiunta in testa alla lista di sezioni originaria, quindi verrà processata all'iterazione successiva, mentre la prima sezione ottenuta viene passata al metodo `scan` e poi aggiunta in coda alla lista di supporto. Al termine dell'iterazione, cioè quando la lista delle sezioni è vuota, la lista di supporto conterrà tutte le sezioni del testo, in ordine e divise correttamente. Quindi diviene la nuova lista delle sezioni memorizzata come variabile d'istanza della classe.

4.1.2 Il metodo `scanSection`

Questo metodo (2) prende in input una `TextSection` e cerca di individuare i campi presenti nella stessa. Per fare questo utilizza la lista `knownFieldsList` che era stata ottenuta dalla classe `DocumentFieldList`, che contiene tutti i campi supportati (si veda a tal proposito l'appendice B. Per ogni elemento di questa lista, che è un istanza di `DocumentField` viene chiamato il metodo `find(String)` dell'elemento stesso, avviando

Algorithm 1 Algoritmo per il metodo scan

```
1:  function SCAN
2:      supportList ← lista di TextSection
3:      while sectionList is not empty do
4:          currentSection ← sectionList.removeFirst()
5:          if currentSection.isModified() then
6:              splitIndex ← currentSection.findSplitIndex()
7:              if splitIndex ≥ 0 then
8:                  splitList ← currentSection.split(splitIndex)
9:                  currentSection ← splitList.removeFirst()
10:                 aggiungi in testa a sectionList tutti gli elementi
                                                    rimanenti di splitList
11:             end if
12:             scanSection(currentSection)
13:             currentSection.freezeText()
14:         end if
15:         supportList.addLast(currentSection)
16:     end while
17:     sectionList ← supportList
18: end function
```

la ricerca di quel particolare campo sulla stringa che costituisce il testo. Se il valore ritornato da questa chiamata non è `[-1,-1]` allora le coppie di indici ottenute vengono usate per selezionare le sezioni di testo opportune. Tali sezioni vengono quindi inserite nell'archivio, e all'interno della stringa su cui si effettua la ricerca (che è una copia dell'originale) vengono sostituite da una serie di $\sim$, in modo da evitare che la stessa sezione di testo venga presa in considerazione nelle scansioni successive per gli altri campi. Così si evita che ad una stessa parola siano associati due significati differenti.

Tutta la procedura appena descritta viene eseguita solo se il cursore visibile dell'editor non è contenuto nella sezione (riga 2 dell'algoritmo 2), altrimenti possono verificarsi situazioni di errore. Ad esempio il notaio sta inserendo l'indirizzo "Via Roma, 303" mentre il metodo `scanSection(TextSection)` è in esecuzione sulla sezione in cui è contenuto il cursore di inserimento dell'editor. Se il metodo `find(String)` del `DocumentField` "Indirizzo" viene chiamato quando l'utente ha scritto "Via Roma, 3" tale stringa verrà identificata come indirizzo, anche se non è quello che l'utente intendeva inserire, dato che doveva ancora finire di digitarlo. Una volta terminata la digitazione, alla successiva scansione della sezione verrà riconosciuto l'indirizzo corretto, e quindi in archivio si avranno due indirizzi: "Via Roma, 3" e "Via Roma, 303", il primo dei quali è chiaramente errato. Per evitare queste situazioni quindi la scansione non viene eseguita sulle sezioni in corso di modifica.

Algorithm 2 Algoritmo per il metodo scanSection

```
1:  function scanSection(section)
2:      if section contiene il cursore visibile then
3:          return
4:      end if
5:      toScan ← section.getString()
6:      for all documentField in knownFieldsList do
7:          fieldIndexes ← documentField.find(toScan)
8:          for all coppia di indici indexCouple in fieldIndexes do
9:              if gli indici sono validi then
10:                 selectedField ← seleziona la porzione di testo
                                    compresa fra l'indice di inizio e di fine
11:                 inserisci selectedField nell'archivio, con chiave
                                                documentField.getName()
12:                 sostituisci la stringa appena trovata con una serie di
                                                "∼" di pari lunghezza
13:             end if
14:         end for
15:     end for
16: end function
```

4.2 METODI DELLA CLASSE `textsection`

Come esposto nella sezione 3.3 la classe `TextSection` rappresenta l'unità base su cui si svolge la ricerca. Deve fornire la possibilità di controllare se è stata modificata dall'ultima scansione, in modo da evitare scansioni inutili, e di trovare un eventuale punto di fine frase. La presenza di tale punto infatti indica la necessità di dividere la sezione in due sezioni più piccole. Infine è neccessario il metodo che opera questa divisione, restituendo due sezioni distinte, separate dal punto di fine frase trovato con `findSplitIndex()`.

In tutti gli algoritmi seguenti sono utilizzati le seguenti variabili d'esemplare della classe `TextSection`:

- `text` : di tipo `XTextRange`, possiede il metodo `getString` che restituisce la stringa correntemente contenuta nella sezione
- `lastSearchedText` : è una stringa che viene usata per memorizzare il testo su cui è stata effettuata l'ultima ricerca

4.2.1 Il metodi `freezeText` e `isModified`

Questi due semplici metodi svolgono un ruolo cruciale nel procedimento di ricerca. Senza di essi infatti sarebbe necessario scansionare ripetutamente l'intero documento.

Il metodo `freezeText()` (3) effettua uno *snapshot* del testo contenuto all'interno della sezione, memorizzandolo in un campo d'esemplare della sezione. In questo modo la verifica dell'avvenuta modifica del testo è altrettanto semplice. Infatti basta confrontare questo campo d'esemplare col testo contenuto nella sezione al momento della chiamata a `isModified()` (4).

Tutto questo funziona se il metodo `freezeText` viene chiamato subito dopo aver effettuato la scansione della sezione, come avviene nel metodo `scanSection(TextSection)` (sezione 4.1.2). In tal modo il testo memorizzato è sempre l'ultimo su cui è stata fatta la scansione.

Algorithm 3 Algoritmo per il metodo freezeText

```
1: function FREEZETEXT
2:     lastSearchedText ← text.getString()
3: end function
```

Algorithm 4 Algoritmo per il metodo isModified

```
1: function ISMODIFIED
2:     return lastSearchedText ≠ text.getString()
3: end function
```

4.2.2 Il metodo `findSplitIndex`

Qui ci concentriamo sull'analisi del metodo `findSplitIndex`. Il suo compito è cercare la presenza di un segno di punteggiatura che termina una frase all'interno della sezione. Se lo trova ne ritorna l'indice all'interno della stringa che rappresenta il testo, altrimenti ritorna -1.

L'idea di base dell'algoritmo è cercare i segni di punteggiatura specificati in `pointPattern` (ovvero !,? e .) e ogni volta che se ne trova uno controllare se è valido. Per fare questo si utilizza il metodo `matches` degli elementi di `notPointList`. Se il punto viene riconosciuto come non valido (ad esempio fa parte dell'abbreviazione "sig.") allora si procede con la ricerca, altrimenti l'indice del punto trovato viene ritornato, per essere utilizzato come punto di spezzamento fra due sezioni.

Algorithm 5 Algoritmo per il metodo findSplitIndex: restituisce l'indice del primo punto di fine frase valido. Se non è presente nessun segno di punteggiatura valido, ritorna -1.

1: **function** FINDSPLITINDEX

- *pointPattern*: oggetto di tipo `Pattern`
- *notPoint*: oggetto di tipo `NotEndOfSentenceRegex`. Possiede il metodo `matches(String, int)` che verifica se la porzione di testo attorno all'indice dato rende il punto di fine frase o no. (Alg. 7)
- *notPointList*: lista che contiene tutti i `NotEndOfSentenceRegex` supportati.

2: *currentText* ← text.getString() // stringa su cui fare la ricerca
3: **while** *pointPattern* viene trovato **do**
4: *index* ← indice dell'occorrenza di *pointPattern* appena trovata
5: *matchedSomethig* ←`false`
6: **for all** *notPoint* **in** *notPointList* **do**
7: **if** *notPoint*.matches(*currentText*, *index*) **then**
8: *matchedSomethig* ← `true`
9: **end if**
10: **end for**
11: **if** !*matchedSomethig* **then**
12: **return** *index*
13: **end if**
14: **end while**
15: **return** -1
16: **end function**

4.2.3 Il metodo `split`

Questo metodo è fondamentale, in quanto consente di mantenere la corretta divisione in sezioni man mano che l'utente inserisce testo. Sfrutta le proprietà degli `XTextCursor` di OpenOffice per selezionare il testo da trasformare in sezione. Infatti i cursori oltre a essere mossi possono essere espansi, in questa maniera si seleziona del testo (in questo caso fino al punto di fine frase, e poi di lì in poi) e se ne fa una nuova sezione. Le due nuove sezioni vengono inserite in una lista, che verrà poi inserita dal chiamante (la classe `SectionScanner`) al posto della sezione appena splittata nella lista di tutte le sezioni.

Algorithm 6 Algoritmo per il metodo split

```
1:  function SPLIT(splitIndex)
2:      splitList ← lista di sezioni
3:      if splitIndex = -1 then
4:          add this section to splitList
5:          return splitList;
6:      end if
7:      middle ← il cursore posizionato a splitIndex
8:      crea un cursore first posizionato all'inizio della sezione e
                    muovilo fino al cursore middle selezionando il testo
9:      crea una nuova TextSection col testo del cursore first
10:     aggiungi la TextSection appena ottenuta in splitList
11:     crea un cursore second posizionato alla fine della sezione e
                    muovilo fino al cursore middle selezionando il testo
12:     crea una nuova TextSection col testo del cursore second
13:     aggiungi la TextSection appena ottenuta in splitList
14:     return splitList;
15: end function
```

4.3 METODI DI CLASSI ACCESSORIE

Tutta la struttura di ricerca presentata finora richiede delle classi accessorie, come ad esempio `NotEndOfSentenceRegex`, utilizzata nell'algoritmo 5. Il metodo fondamentale di questa classe, che rappresenta i casi particolari in cui un segno di punteggiatura non chiude una frase, è presentato nell'algoritmo 7.

Gli oggetti di tipo `NotEndOfSentenceRegex` hanno tre attiubuti principali:

1. preOffset
2. postOffset
3. pattern

Algorithm 7 Algoritmo per il metodo matches

```
1: function MATCHES(text, center)
2:     if center + preOffset < 0 then                    // Per evitare
                                                          IndexOutOfBoundException
3:         toMatch = text.substring(0, center + postOffset);
4:         return pattern.matcher(toMatch).matches();
5:     end if
6:     toMatch = text.substring(center + preOffset, center + postOff-
   set);
7:     return pattern.matcher(toMatch).matches();
8: end function
```

`preOffset` e `postOffset` sono due interi che indicano come deve essere posizionato il `pattern` rispetto al segno di punteggiatura in esame. Ad esempio, il punto che termina l'abbreviazione "Sig." non indica la fine di una frase, quindi esiste un NotEndOfSentenceRegex che lo descrive, e che nel file di configurazione (vedere B.2) è

```
<NEOS_Regex regex="[Ss]ig" preOffset="-3" postOffset="0" />
```

Il significato dei parametri è che il pattern `[Ss]ig` deve essere composto esattamente dai 3 caratteri prima (il preOffset) del punto in esame. Siccome questo particolare caso non comprende il punto stesso (a differenza ad esempio delle cifre decimali, come "3.4") allora il postOffset sarà 0.

5 ANALISI DELLE PRESTAZIONI

In questo capitolo si traggono le conclusioni sulle prestazioni della ricerca nel testo, sia dal punto di vista della complessità computazionale che da quello dell'accuratezza dei risultati. Nella sezione 5.1 si tratta della complessità temporale dell'algoritmo di ricerca (5.1.2) e del metodo di ricerca delle espressioni regolari della libreria standard di Java (5.1.1).

5.1 COMPLESSITÀ COMPUTAZIONALE DELLA RICERCA

In questa sezione si esamina la complessità di tempo del metodo di ricerca utilizzato per trovare i campi nel testo.

5.1.1 Complessità del match nell'implementazione Java delle espressioni regolari

Questa è la parte di codice che viene eseguita più frequentemente, quindi quella che incide in maggior misura sulle prestazioni. Per motivi di tempo e affidabilità il programma sfrutta l'implementazione della ricerca tramite espressioni regolari fornita da java.

Il tempo impiegato durante l'utilizzo delle espressioni regolari si suddivide sostanzialmente in due fasi: la compilazione e il tentativo di match. La compilazione consiste nella trasformazione dell'espressione regolare in un automa a stati finiti deterministico (DFA) che verrà poi usato nella fase di match, durante la quale viene effettivamente letta la stringa in input.

Per ridurre il tempo totale di esecuzione la compilazione dell'espressione regolare viene fatta una volta sola, al momento del caricamento dal file di configurazione dei campi supportati (descritto in B.1) all'inizio dell'esecuzione del programma. La lista di `DocumentField` quindi contiene espressioni regolari già compilate, di conseguenza il tempo di compilazione non incide sul tempo di ricerca.

L'analisi è più complessa per quanto riguarda l'efficienza dell'implementazione java delle espressioni regolari. Infatti non dipende solamente dall'input, ma anche dalla struttura stessa dell'espressione regolare. Per alcune espressioni il tempo di ricerca è lineare, mentre per alcuni casi particolari è addirittura esponenziale. Parte di questa dipendenza risiede proprio nell'implementazione usata in Java, che si basa su un approccio simile a quella di Perl, Python, PHP e molti altri linguaggi, ovvero l'implementazione *backtracking* ricorsiva, che è una maniera di simulare un automa a stati finiti non deterministico (NFA). Il vantaggio risiede nella flessibilità e nella disponibilità di funzionalità aggiuntive, che permettono di identificare alcuni linguaggi che non

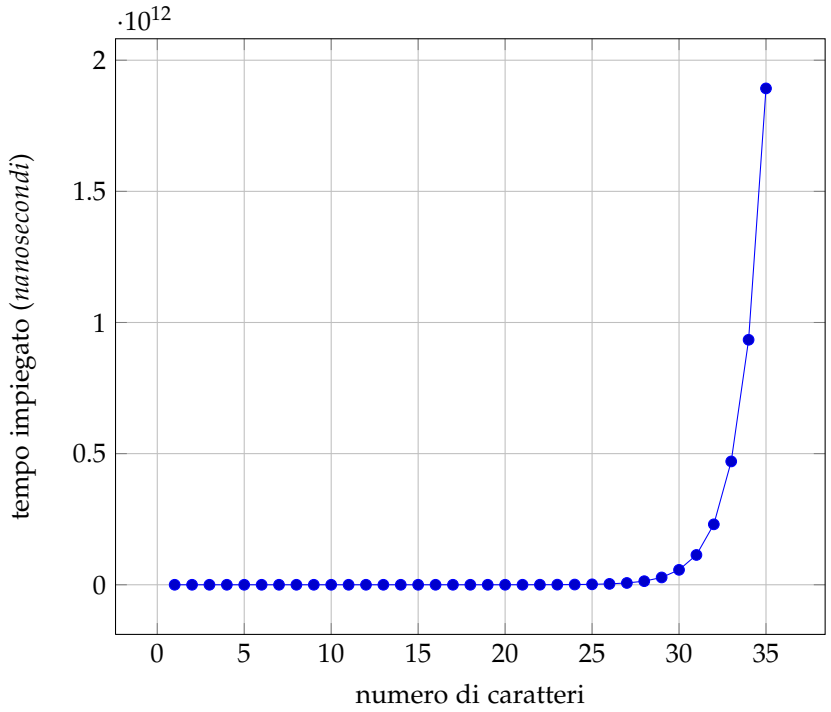

Figura 13.: Numero di caratteri contro tempo necessario per il match per l'espressione regolare $x?^n x^n$ contro il testo x^n, con n che va da 1 a 35. Per una stringa composta da 35 x il tempo impiegato è oltre 30 minuti.

sono propriamente regolari (e quindi oltre le capacità di un NFA), ma ha il problema che esistono alcuni casi "patologici". Un'alternativa è l'implementazione proposta da Ken Thompson [11] e che viene usata in programmi come `awk`, `grep` e `sed` e che non ha problemi di questo tipo.

Un esempio di caso patologico per l'implementazione *backtracking* è costituito dall'espressione

$$x?^n x^n$$

con come input stringhe del tipo x^n. Gli esponenti sono un'abbreviazione per la ripetizione della stringa, quindi $x?^2 x^2$ sta per $x?x?xx$. È evidente che la stringa x^n appartiene al linguaggio descritto dall'espressione regolare, tuttavia in tale caso il tempo di esecuzione è esponenziale. Questo perché l'approccio *backtracking* quando si trova di fronte l'operatore ? (ovvero 0 o 1 corrispondenze) prova prima a ottenere una corrispondenza e poi zero. Dato che l'operazione deve essere compiuta n volte, decidendo ogni volta per "zero corrispondenze", il numero totale di confronti è 2^n, senza contare la seconda parte dell'espressione regolare, che invece farà solamente n confronti. Di conseguenza la complessità di tempo è $O(2^n)$.

Che l'andamento in questo caso sia esponenziale lo conferma anche una verifica sperimentale (fig 13). [1]

1 Test eseguito su Intel core 2 duo, 2,53 GHz, 4 Gb RAM, sistema Debian GNU/Linux

Questo e casi simili di comportamento esponenziale sono fortunatamente rari e non compaiono nelle espressioni regolari utilizzate dal programma. Le espressioni utilizzate non solo evitano forme patologiche come quelle esposte, ma non sfruttano le estensioni aggiunte dall'implementazione Java per riconoscere linguaggi anche non regolari, come ad esempio *lookahead* e *lookbehind*, che aumentano significativamente il tempo di esecuzione.

Quindi le espressioni regolari usate esibiscono un comportamento lineare nel tempo di ricerca.

5.1.2 Complessità temporale della ricerca dei campi

Prima di procedere all'analisi della complessità computazionale dei vari algoritmi che compongono la ricerca, sono qui riassunti brevemente i passi fondamentali.

1. Ottenere la prima sezione del testo
   a) Cercare il primo segno di punteggiatura di fine frase
   b) Se un punto di fine frase è presente, dividere in due la sezione
2. Se la sezione è stata modificata, effettuarne la scansione
   a) Per ogni campo supportato, cercarne eventuali occorrenze
3. Ottenere la sezione successiva come da punto 1., continuando a ripetere

La parte cruciale della ricerca, sia dal punto di vista dell'accuratezza che del tempo impiegato, è la scansione di una singola sezione.

Come visto in 4.1.2 l'algoritmo utilizza due cicli annidati per effettuare la scansione. Il ciclo più esterno itera su tutti i `DocumentField` conosciuti, che sono in numero fisso, indipendente dalle dimensioni dell'input. Questo ciclo si traduce quindi in un termine costante nell'espressione della complessità. All'interno di questo ciclo viene chiamato il metodo `find()` del `DocumentField` corrente (la cui complessità sarà analizzata in seguito) e il ciclo su tutte le occorrenze del campo trovate dal metodo `find()` per inserirle nell'archivio. Tale ciclo sarà ripetuto un numero di volte nel peggiore dei casi proporzionale all'input.

Di seguito è riportata l'analisi della complessità del metodo find per le due differenti implementazioni usate: quella della classe `RegexBasedDocumentField` e `SetBasedDocumentField`.

REGEX BASED DOCUMENT FIELD Questo metodo utilizza un'espressione regolare già caricata dal file di configurazione e compilata. Utilizzando il metodo di ricerca della classe `java.util.regex.Pattern` trova tutte le occorrenze del campo all'interno del testo che gli viene fornito come parametro (ovvero il testo della sezione) e ne memorizza gli indici di inizio e fine in una lista che viene restituita al chiamante. Come visto in 5.1.1 tale tempo di ricerca è lineare in assenza di espressioni regolari patologiche, come in questo caso.

SET BASED DOCUMENT FIELD La procedura usata qui è del tutto simile alla precedente, con la differenza che la ricerca tramite espressione regolare restituisce solamente dei candidati. Successivamente è necessario verificare che questi candidati siano contenuti negli insiemi di parole conosciuti. La struttura scelta per implementare questi insiemi (si veda l'appendice A) è `java.util.HashSet`, che ha un tempo di accesso che è in media $O(1)$. Così il tempo di ricerca sul testo passato come parametro anche in questo caso è lineare rispetto alle dimensioni dell'input.

È qui riportato un elenco riassuntivo delle varie complessità coinvolte nella ricerca. Tutte le complessità sono da intendersi di tempo rispetto alla dimensione n dell'input in caratteri.

1. Trovare il punto di fine frase: $O(n)$
2. Dividere in due la sezione: $O(1)$
3. Scansione di una sezione: $O(n)$ (vedere più sotto, e con riferimento all'algoritmo 2)
   a) ciclo esterno dell'algoritmo (riga 6 dell'algoritmo 2): $O(1)$ rispetto alle dimensioni dell'input, perché viene eseguito sempre un numero fissato di volte pari a d, ovvero il numero di campi conosciuti.
   b) metodo `find()` : $O(n)$ sia per i campi basati su espressione regolare che sull'appartenenza a un insieme.

La complessità totale del meccanismo di ricerca è quindi data da

$$\sum_{i=0}^{d} (O(n) + O(n)) = 2dO(n) = O(n)$$

dove d è il numero di `DocumentField` conosciuti e n è la dimensione della stringa che costituisce la sezione. All'interno della sommatoria il primo termine indica la complessità del metodo `find()` del `DocumentField`, e il secondo termine è il caso peggiore del ciclo più interno, che itera su tutte le occorrenza trovate del campo.

DATI SPERIMENTALI Di seguito sono presentati alcuni risultati ottenuti sperimentalmente per confermare l'analisi fatta in precedenza.

Durante la ricerca il dato significativo sul tempo di esecuzione è fornito dal tempo di scansione di una sezione. Infatti il tempo per la scansione dell'intero documento dipende da quante e quali sezioni sono state modificate e necessitano di scansione, e quindi non fornisce un buon indicatore delle prestazioni dell'algoritmo. Inoltre il documento viene visto come una semplice collezione di sezioni, quindi il dato che sta veramente al centro della procedura è il tempo di scansione di una singola sezione.

La procedura utilizzata per i test è consistita nella scelta di alcune frasi, inserite un numero sempre maggiore di volte nel testo, ma non contenenti un segno di punteggiatura di fine frase. In questo modo il

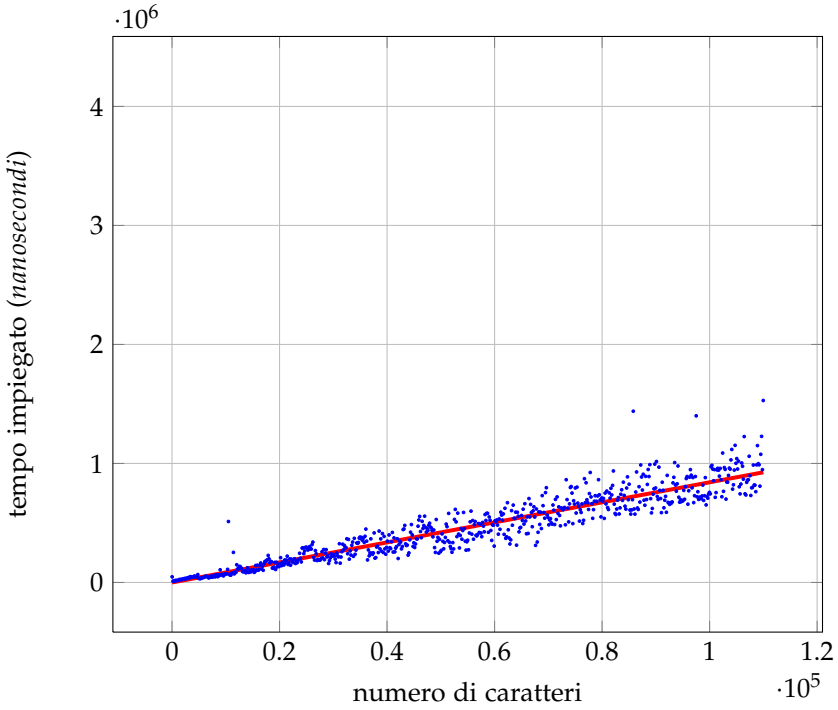

Figura 14.: Tempo impiegato nella scansione di una testo composto da ripetizioni delle stringa "il Signor Mario RossI". I punti · rappresentano le misure e — è la retta di regressione lineare calcolata col metodo dei minimi quadrati. Grafico ottenuto con 5000 punti dati

programma vede il testo come formato da un'unica sezione ed esegue la scansione di volta in volta. [2]

Il primo test utilizza la stringa

"il Signor Mario RossI "

con la I maiuscola finale. Questo non è un nome per l'espressione regolare che li descrive, quindi, essendo solo l'ultimo carattere a non corrispondere alla forma corretta, questa stringa costituisce un caso problematico. Tale stringa è stata scelta perché (a parte l'ultimo carattere) ricade nella categoria dei nomi propri, che sono descritti dall'espressione regolare più complessa utilizzata dal programma. Inoltre questa espressione regolare fa uso di una funzionalità dell'implementazione Java che potenzialmente può rallentare l'esecuzione, ovvero la selezione di un sottogruppo del match per eliminare dalla selezione la parte "Signor". Il grafico con stringhe di questo tipo di differente lunghezza è mostrato in figura 14.

Un altro test è stato svolto con la stringa

"In Roma, nel mio studio, Via del Corso n.303 Davanti a me dottor Maria Emanuela VESCI, Notaio in Roma, iscritto nel Ruolo dei Distretti Notarili Riuniti di Roma, Velletri e Civitavecchia, senza l'assistenza dei testimoni avendovi i comparenti, d'accordo fra loro e con il mio consenso, rinunziato ".

2 Test eseguiti su Intel Core 2 Duo 2.53 GHz, RAM 4 Gb, sistema Debian GNU/Linux

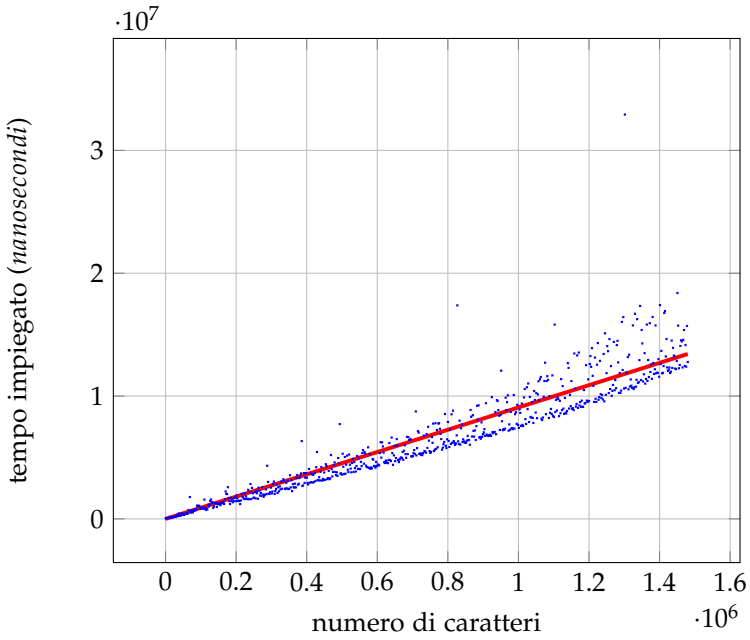

Figura 15.: Tempo impiegato nella scansione di un testo composto da ripetizioni delle stringa "In Roma, nel mio studio, Via del Corso n.303 Davanti a me dottor Maria Emanuela VESCI, Notaio in Roma, iscritto nel Ruolo dei Distretti Notarili Riuniti di Roma, Velletri e Civitavecchia, senza l'assistenza dei testimoni avendovi i comparenti, d'accordo fra loro e con il mio consenso, rinunziato". La misure dettano la posizione dei punti ·, mentre — rappresenta la retta di regressione lineare calcolata col metodo dei minimi quadrati. I picchi dei dati sono dovuti all'intervento del *garbage collector* che introduce un ritardo e quindi rumore nelle misure. Grafico ottenuto con 5000 misure.

Tale stringa è stata presa da una atto reale, ed è formata da una sola sezione. Inoltre contiene vari campi: un indirizzo, un nome e alcune città. Anche in questo caso il tempo di esecuzione della ricerca su testi composti da ripetizioni di questa stringa è lineare, come si evince dal grafico in figura 15.

Nel caso di testi che non contengono alcun campo, come ad esempio

"questa è una frase senza alcun campo "

il tempo di esecuzione della ricerca è ancora lineare. La figura 16 ne mostra l'andamento.

5.2 ANALISI DELL'ACCURATEZZA

In questa sezione ci si occupa di analizzare l'accuratezza dell'algoritmo di ricerca nel trovare i campi nel testo.

I test sono stati eseguiti su alcuni atti notarili trovati in rete (una decina in tutto, vista la difficoltà nel reperirli) e sono stati presi alcuni dei campi più significativi e presenti con maggior frequenza per avere

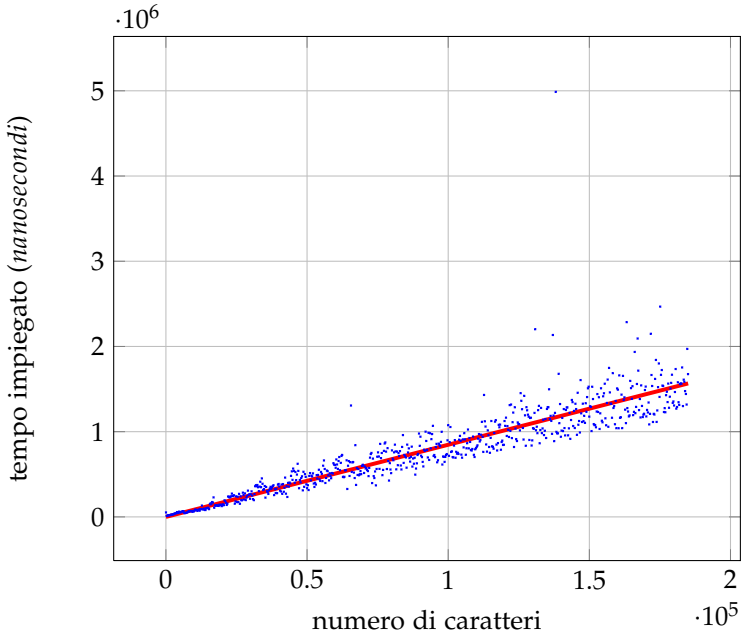

Figura 16.: Tempo impiegato nella scasione di una testo composto da ripetizioni delle stringa "questa è una frase senza alcun campo". I punti · sono definiti dalle misure, mentre — è la retta di regressione lineare calcolata col metodo dei minimi quadrati. Grafico ottenuto con 5000 misure.

dei risultati validi. I campi di cui è stato tenuto conto in questa analisi sono:

- Nome
- Data
- Indirizzo
- Codice fiscale
- Partita IVA
- Città

Gli altri campi non sono altrettanto usati, quindi non costituiscono un buon campione. È da notare in ogni caso che i campi "Numero di repertorio" e "Numero di raccolta", che costituiscono un identificativo dell'atto, sono sempre stati identificati correttamente nell'intestazione dei documenti utilizzati per i test. Per ogni atto è stata effettuata una scansione su tutto il documento, procedendo quindi al conteggio dei campi identificati correttamente e di quelli non identificati o identificati erroneamente. La tabella 2 riassume i risultati del test di accuratezza.

Dai risultati esposti nella tabella si nota che codici fiscali e partite IVA sono sempre trovati correttamente. Questi due campi ricadono

Campo	Percentuale occorrenze identificate
Nome	94, 11%
Data	95, 74%
Indirizzo	86, 95%
Codice fiscale	100%
Partita IVA	100%
Città	91, 66%

Tabella 2.: Occorrenze identificate correttamente dei campi maggiormente utilizzati negli atti notarili.

nella categoria di quelli trovati tramite le espressioni regolari e, avendo entrambi una struttura ben definita, vengono descritti in maniera molto precisa da un'espressione regolare semplice.

Per quanto riguarda i risultati ottenuti per i campi "Città" e "Nome" la mancata identificazione di tutte le occorrenze è dovuta all'assenza della parola (ad esempio della città) nell'elenco dei valori possibili per il campo. Infatti entrambi sono campi basati sull'appartenenza a un insieme, e aggiungendo ai file di dati le parole mancanti queste vengono identificate correttamente.

Il discorso è diverso con i campi "Data" e "Indirizzo", specialmente con quest'ultimo. Entrambi questi campi sono riconosciuti tramite espressioni regolari, tuttavia la loro struttura è soggetta a importanti variazioni, non facilmente prevedibili. Un campo che nel testo si presenta in una forma diversa da quella prevista nel file di configurazione `DocumentFields.xml` non viene identificato. La soluzione per questo problema è modificare l'espressione regolare in modo da adattarla al nuovo caso. Il fatto che tale espressione sia scritta in un file di configurazione e non nel codice ne rende più semplice e sicura la modifica, infatti un errore non compromette l'intero meccanismo di ricerca, ma al più impedisce di trovare correttamente i campi corrispondenti a quell'espressione.

Infine, per quanto concerne i campi non inclusi in quest'analisi a causa del fatto che non sono spesso presenti, la loro correttezza è testata attraverso lo strumento di testing JUnit. [3] In particolare i campi "Numero di repertorio" e "Numero raccolta" fanno parte dell'intestazione di ogni atto e sono stati trovati tramite espressione regolare in ogni documento utilizzato. Talvolta questi due campi sono presenti anche in altri punti dell'atto, come riferimento ad altri documenti, e siccome la loro forma in questo caso varia molto, non sempre vengono trovati correttamente, ma ai fini della compilazione della modulistica le occorrenze rilevanti sono quelle che compaiono nell'intestazione.

3 `www.junit.org`

6 CONCLUSIONI

Il progetto ELaw ha come obiettivo la realizzazione di uno studio notarle informatizzato. In particolare questo studio include un applicativo per la videoscrittura che fornisce dei metodi per aumentare la produttività, diminuendo il tempo necessario a trascrivere le informazioni da atti già redatti a moduli che devono essere inviati agli organi competenti. Per raggiungere questo scopo si è utilizzato come base il Word Processor OpenOffice, per il quale è stata sviluppata un'estensione che unisce in un unico passo la stesura dell'atto e la compilazione dei moduli.

Infatti quello che fa questa estensione è cercare delle parole significative (campi) all'interno del documento durante la sua stesura e memorizzarle in un archivio per usarle successivamente nella produzione dei file che devono essere inviati.

In questo elaborato ci si è occupati della parte relativa alla ricerca dei campi nel testo. L'idea fondamentale su cui si poggia l'intero meccanismo è che la ricerca viene fatta durante la stesura del documento. Risulta quindi necessario analizzarlo periodicamente per trovare eventuali nuovi campi. Per evitare sprechi di risorse e tempo con una scansione completa ogni volta, il documento viene diviso in frasi, e rappresentato come una lista di frasi. Un ciclo di ricerca consiste quindi nello scorrere tale lista e nello scansionare solo le frasi che hanno subito modifiche dall'ultima scansione.

Più in particolare la scansione delle frasi si basa sulla divisione dei campi in due categorie: quelli che possono essere identificati da espressioni regolari e quelli che, oltre ad avere una struttura identificata da un'espressione regolare, devono soddisfare l'appartenenza a un insieme di parole (come ad esempio le città). Sono quindi fondamentali le espressioni regolari nella ricerca, scelte perché forniscono un metodo flessibile ed efficiente per descrivere e cercare stringhe in un testo. L'intera procedura di ricerca ha una complessità temporale rispetto alle dimensioni dell'input che è $O(n)$. I test effettuati mostrano inoltre che l'accuratezza della ricerca supera il 90%.

L'approccio modulare scelto fa sì che il programma sia facilmente estendibile: nuovi campi possono essere aggiunti semplicemente modificando il file di configurazione

opportuno e, se un campo non dovesse ricadere nelle categorie precedentemente descritte, una nuova categoria può essere aggiunta semplicemente creando una sottoclasse di `DocumentField`.

Infine sono previsti ulteriori sviluppi per quanto riguarda la ricerca nel testo, principalmente in due direzioni: l'apprendimento automatico di nuovi campi a partire dalle indicazioni dell'utente e la capacità di trovare le relazioni tra campi diversi.

A SCELTA DELLA STRUTTURA DI MEMORIZZAZIONE DEGLI ELENCHI DI PAROLE CONOSCIUTE: TEST

In questa appendice sono descritti i test che hanno portato alla scelta della struttura dati utilizzata dalla class `SetBasedDocumentField`.

Durante lo sviluppo della classe `SetBasedDocumentField` si è posto il problema di quale struttura utilizzare per memorizzare l'insieme di parole possibili. L'obiettivo è di avere una struttura efficiente dal punto di vista del tempo di ricerca e possibilmente piccola come occupazione di spazio. Le strutture prese in esame sono 4:

TRIE : implementato nella classe `search.dataStructures.TrieSet` memorizza le parole appartenenti all'insieme in un trie.

BUFFER : Carica in RAM solo un buffer del file di dati contenente le parole.

PATTERN : Costruisce un'espressione regolare del tipo

```
(parola1|parola2|parola3|...)
```

che enumera tutte le possibili parole.

HASHTABLE : memorizza tutte le parole in una hashtable (più precisamente un'istanza di `java.util.HashSet`).

Alcuni grafici di confronto sono presentati in figura 17 e 18. Il primo grafico è stato ottenuto creando un insieme di stringhe numeriche, usando le diverse implementazioni in esame, e cercando un elemento che non fosse presente. Con la stessa procedura sono stati creati anche degli insiemi di cui si è misurata l'occupazione di spazio riportata nel secondo diagramma.

Dopo i primi test la struttura a buffer è stata subito scartata, infatti, pur avendo un'occupazione di spazio costante, è più lenta di almeno due ordini di grandezza rispetto alle altre (figura 17 linea —■—).

Per quanto riguarda le altre strutture dal grafico 17 risulta che trie e hastable hanno tempi di ricerca simili, mentre l'insieme costruito attraverso pattern è più lento. Inoltre la velocità di quest'ultimo è dipendente dalla taglia dell'insieme, mentre per hashtable e trie il tempo è circa costante, indipendentemente dal numero di parole. Questo porta ad escludere l'insieme basato su pattern, essendo la velocità la prima figura di merito.

Tra trie e hashtable, che effettuano la ricerca in tempi comparabili, la scelta è ricaduta su hashtable per due motivi. Il primo è che hashtable occupa meno spazio, come si vede anche dal grafico 19. Inoltre la classe `java.util.HashSet` fa parte della libreria standard di java, quindi è

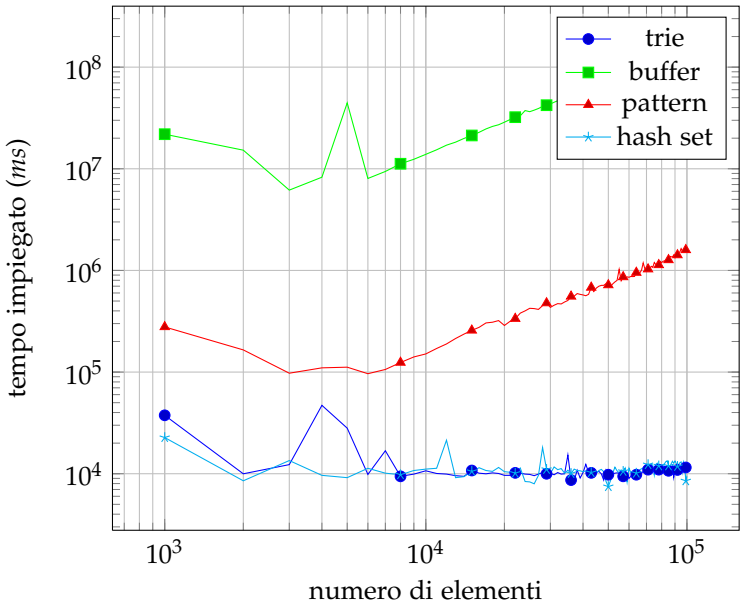

Figura 17.: Numero di elementi contro tempo nella ricerca, confronto varie strutture dati

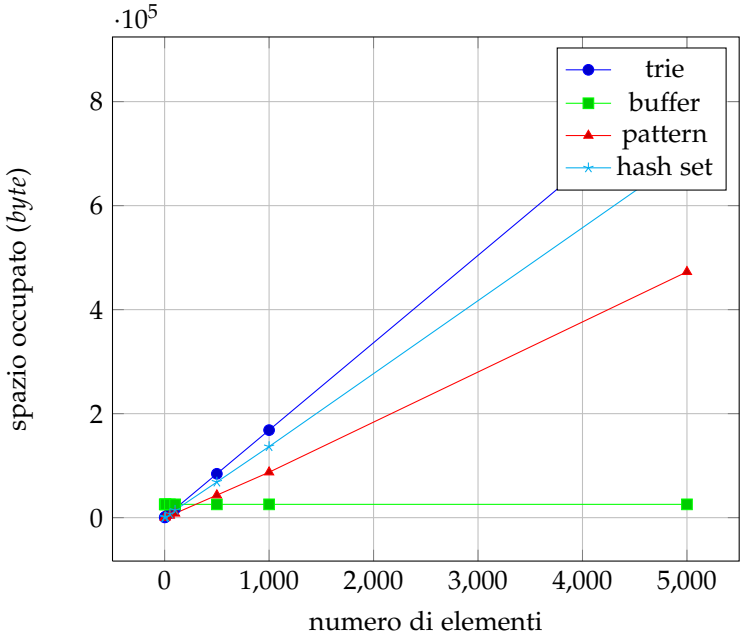

Figura 18.: Numero di elementi contro occupazione di spazio, confronto varie strutture dati

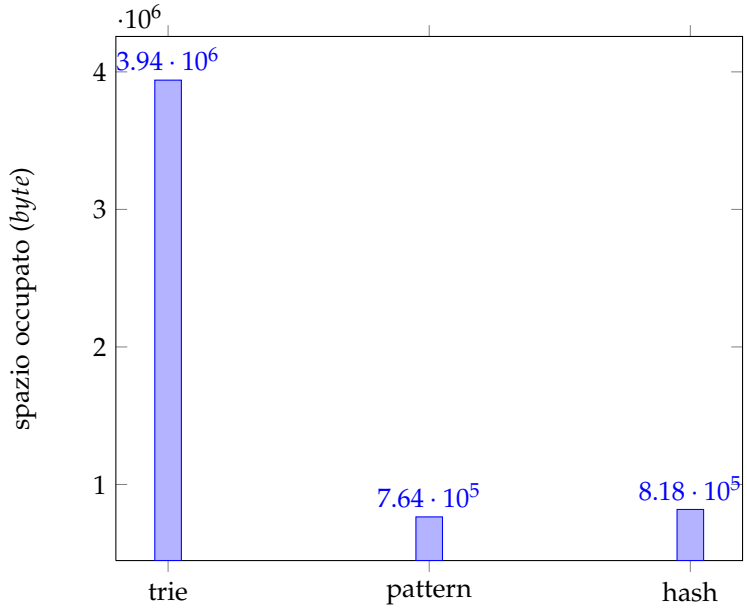

Figura 19.: Spazio occupato da strutture dati differenti per dati pratici

più testata e meno soggetta a bug rispetto all'implementazione del trie fatta ad hoc per il progetto.

Un confronto fra trie, pattern e hashtable fatto con dati che poi vengono realmente usati dall'applicazione è presentato nel listato 4, che è l'output di un programma di test sulla lista dei comuni italiani. Le misure effettuate sono le seguenti:

- riga 1 : lo spazio in *byte* occupato dalle tre strutture, riempite con i nomi dei comuni italiani.
- righe 8 e 15 : il tempo per cercare due differenti comuni presenti nell'insieme, rispettivamente il primo della lista e uno a metà.
- riga 22 : è il tempo impiegato per verificare che una stringa (*Pizzighettoni*) non è contenuta nell'insieme

Misure più significative dei tempi sono però date da

- riga 29 : la media dei tempi di ricerca per tutte le stringhe contenute nell'insieme. Valore ottenuto effettuando la ricerca per ogni stringa contenuta nell'insieme nelle tre differenti implementazioni e facendo poi la media aritmetica dei tempi.
- riga 36 : simile alla misura precedente, è la media dei tempi per la ricerca di stringhe non appartenenti all'insieme, ottenuta cercando nelle strutture dati stringhe di caratteri casuali di lunghezza casuale (fra 1 e 20).

È evidente che mentre hashtable e pattern sono simili come occupazione di spazio, quest'ultimo è nettamente più lento. Hashtable e trie sono paragonabili come tempi di ricerca, ma il trie occupa molto spazio in più.

```
Space for "comuni" (bytes)

 771032 :       class search.dataStructures.HashBasedSet
3714464 :       class search.dataStructures.TrieSet
 722800 :       class search.dataStructures.PatternBasedSet


Time for a string (Pizzighettone) contained in "comuni" (nanoseconds)

 20393  :       class search.dataStructures.HashBasedSet
 16832  :       class search.dataStructures.TrieSet
703511  :       class search.dataStructures.PatternBasedSet


Time for a string (Abano Terme) contained in "comuni" (nanoseconds)

 3213   :       class search.dataStructures.HashBasedSet
 3423   :       class search.dataStructures.TrieSet
26400   :       class search.dataStructures.PatternBasedSet


Time for a string not contained in "comuni" (nanoseconds)

  2025  :       class search.dataStructures.HashBasedSet
  5518  :       class search.dataStructures.TrieSet
823847  :       class search.dataStructures.PatternBasedSet


Mean time of execution for strings that are in set

 1466.37:       class search.dataStructures.HashBasedSet
 1597.97:       class search.dataStructures.TrieSet
53079.83:       class search.dataStructures.PatternBasedSet


Mean time of execution for strings that are not in set, 200 samples

 1327.69:       class search.dataStructures.HashBasedSet
  956.48:       class search.dataStructures.TrieSet
99755.01:       class search.dataStructures.PatternBasedSet
```

Listing 4: Risultati dei test per occupazione di spazio e tempo di esecuzione sull'elenco dei comuni italiani. Tutti i tempi sono in nanosecondi.

B FILE DI CONFIGURAZIONE

In questa appendice si parla dei file di configurazione utilizzati dal programma.

Il programma usa dei file di configurazione in formato XML in cui sono memorizzate le descrizioni dei campi da trovare. In tal modo la descrizione dei campi non è *hard coded* e aggiungerne di nuovi non implica la modifica del codice.

B.1 IL FILE DOCUMENTFIELDS.XML

Il file di configurazione principale è il file `documentFields.xml`, riportato parzialmente nel listato 5.

Questo file viene letto al momento dell'avvio del programma per popolare la lista della classe `DocumentFieldList` che viene utilizzata dalla classe `SectionScanner`.

```
<DOCUMENT_FIELDS>

  <REGEX_BASED groupNumber="0">
    <NAME>Indirizzo</NAME>
    <REGEX>
        ([Vv]ia|[Pp]iazza|[Cc]orso)['\s\w^\d]*[,\.]?[\s]?[\d]+
    </REGEX>
  </REGEX_BASED>

  <REGEX_BASED groupNumber="0">
    <NAME>Data</NAME>
    <REGEX>
      (
        (         # find an all numeric dates
          \d?\d     # day
          [-/\.]      # separator
          \d?\d     # month
          [-/\.]      # separator
          \d{2}(\d{2})? # year
        )|
        (         # find a date with the month in word form
          \d?\d\s     # day number
          (                 # all months
            [Gg]en(naio)? | [Ff]eb(braio)?|
            [Mm]ar(zo)? | [Aa]pr(ile)?|
            [Mm]ag(gio)? | [Gg]iu(gno)?|
            [Ll]ug(lio)? | [Aa]go(sto)?|
            [Ss]ett(embre)? | [Oo]tt(obre)?|
            [Nn]ov(embre)? | [Dd]ic(embre)?
          )
          \s\d{2}(\d{2})?          # the year: 2 or 4 digits
```

```
32          )
33        )
34      </REGEX>
35    </REGEX_BASED>
36
37    <SET_BASED groupNumber="0">
38      <NAME>Provincia</NAME>
39      <SELECTING_MASK>
40          ([A-Z][a-z]+)([\s][A-Z][a-z]+)*
41      </SELECTING_MASK>
42      <DATA_FILE>province.dat</DATA_FILE>
43    </SET_BASED>
44
45    <SET_BASED groupNumber="2">
46      <NAME>Nome</NAME>
47      <SELECTING_MASK>
48        (               # GROUP 1 ----------
49          (?: [A-Z][A-Za-z]+ \s )*  # these are the words that have
50          (?:             # not to be included in the field
51            Signor | SIGNOR |
52            Notaio | NOTAIO |
53            Dottor(?:essa)? | DOTTOR(?:essa)? |
54            Dott\.ssa | DOTT\.SSA |
55            Io
56          ) \s
57        )*
58        (           # GROUP 2 ----------
59        (?:  [A-Z][a-z]+  |  [A-Z]+   ) # the first word
60          (?:               # every other word
61            \s (?: \w' )?   # the space preceding any not-first word,
62                          # followed by an optional character
63                          # with a ' , as in D'Anton
64            (?:  [A-Z][a-z]+  |  [A-Z]+  )
65          )*
66        )
67          </SELECTING_MASK>
68      <DATA_FILE>names.dat</DATA_FILE>
69    </SET_BASED>
70
71  </DOCUMENT_FIELDS>
```

Listing 5: File di configurazione (frammenti) `documentFields.xml`. Le espressioni regolari non sono influenzate da spazi e sono commentate tramite il carattere #.

B.2 IL FILE NOTENDOFSENTENCEREGEXES.XML

In questa sezione viene descritto il file di configurazione che contiene le eccezioni da utilizzare nell'identificazione della punteggiatura che conclude una frase.

Come per il file di configurazione dei campi, anche questo ha lo scopo di separare le definizioni delle eccezioni dal codice, in modo che eventuali modifiche e aggiunte non richiedano la riscrittura dello

```
<NEOS_Regex_List>
  <NEOS_Regex regex="[Ss]ig" preOffset="-3" postOffset="0" />
  <NEOS_Regex regex="[Ee]gr" preOffset="-3" postOffset="0" />
  <NEOS_Regex regex="[Ar]rt" preOffset="-3" postOffset="0" />
  <NEOS_Regex regex="\d\.\d" preOffset="-1" postOffset="2" />
  <NEOS_Regex regex="\sn" preOffset="-2" postOffset="0" />
  <NEOS_Regex regex="[A-Z]" preOffset="-1" postOffset="0" />
</NEOS_Regex_List>
```

Listing 6: File di configurazione notEndOfSentenceRegexes.xml

stesso, col conseguente dispendio di tempo per validarne nuovamente la correttezza.

Il file (listato 6) è in formato xml ed elenca tutte le eccezioni che non costituiscono punteggiatura di fine frase di cui il programma è a conoscenza. Tali eccezioni sono riportate nei record `NEOS_Regex` (Not End Of Sentence Regex) e hanno tre attributi:

- regex: è l'espressione regolare contro cui confrontare i caratteri che circondano il segno di punteggiatura in esame.
- preOffset: è il numero di caratteri che vanno presi prima del segno di punteggiatura per essere confrontati con l'espressione regolare.
- postOffset: il numero di caratteri da prendere dopo il segno di punteggiatura che devono essere confrontati.

Tramite questi parametri si riesce a distinguere fra segni di punteggiatura che concludono una frase e segni che invece fanno parte di sigle (come Sig.) o cifre (come 3.14).

BIBLIOGRAFIA

[1] Eric Brill. Automatic grammar induction and parsing free text: a transformation-based approach. In *Proceedings of the workshop on Human Language Technology*, HLT '93, page 237–242, Stroudsburg, PA, USA, 1993. Association for Computational Linguistics.

[2] Davide Brugali, Giuseppe Psaila, and Franco Guidi-Polanco. Xml for e-government: A new approach to e-law categorization and retrieval. In *ICWI*, page 643–650, 2003.

[3] Russ Cox. Regular expression matching can be simple and fast, January 2007.

[4] J. P. Dick. Conceptual retrieval and case law. In *Proceedings of the 1st international conference on Artificial intelligence and law*, ICAIL '87, page 106–115, New York, NY, USA, 1987. ACM.

[5] J.E.F. Friedl. *Mastering regular expressions*. O'Reilly Series. O'Reilly, 2006.

[6] Peter Jakson and Isabelle Mouliner. Natural language processing for online applications. 2007.

[7] Jinsuk Kim, Du-Seok Jin, Kwang-Young Kim, and Ho-Seop Choe. Automatic in-text keyword tagging based on information retrieval. *JIPS*, 5(3):159–166, 2009.

[8] Jochen L. Leidner. Toponym resolution in text: annotation, evaluation and applications of spatial grounding. *SIGIR Forum*, 41(2):124–126, December 2007.

[9] David D. Lewis and Karen Spärck Jones. Natural language processing for information retrieval. *Commun. ACM*, 39(1):92–101, January 1996.

[10] OpenOffice.org. *Developer's Guide*, 2007.

[11] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, June 1968.

[12] Yu-Chieh Wu and Chia-Hui Chang. Efficient text chunking using linear kernel with masked method. *Know.-Based Syst.*, 20(3):209–219, April 2007.

[13] Yu-Chieh Wu, Yue-Shi Lee, and Jie-Chi Yang. Robust and efficient multiclass svm models for phrase pattern recognition. *Pattern Recogn.*, 41(9):2874–2889, September 2008.